como construir una api restfull en nestjs

Aquí tienes una guía **paso a paso** para construir una **API RESTful** en **NestJS** desde cero. Incluye instalación, configuración, buenas prácticas y ejemplos.

**1. Instalar NestJS**

Primero instala el **CLI de NestJS** globalmente:

npm install -g @nestjs/cli

Crea un nuevo proyecto:

nest new my-api

Te pedirá elegir un **gestor de paquetes** (npm o yarn).  
Una vez creado, entra a la carpeta:

cd my-api

Estructura inicial generada:

src/

├── app.controller.ts

├── app.module.ts

├── app.service.ts

└── main.ts

**2. Configurar el servidor**

El archivo main.ts es el punto de entrada.  
Asegúrate de habilitar **CORS** y configurar el puerto:

// src/main.ts

import { NestFactory } from '@nestjs/core';

import { AppModule } from './app.module';

async function bootstrap() {

const app = await NestFactory.create(AppModule);

app.enableCors(); // Habilitar CORS

await app.listen(3000);

console.log('API running on http://localhost:3000');

}

bootstrap();

**3. Crear un módulo**

Cada recurso debe tener su **propio módulo**.

Ejemplo: módulo products:

nest generate module products

# o simplemente

nest g module products

Esto crea:

src/products/

└── products.module.ts

**4. Crear un controlador**

El **controlador** define las rutas HTTP (GET, POST, PUT, DELETE).

nest g controller products

Esto genera products.controller.ts:

import { Controller, Get, Post, Put, Delete, Param, Body } from '@nestjs/common';

@Controller('products') // Base URL: /products

export class ProductsController {

private products = [];

@Get()

getAll() {

return this.products;

}

@Get(':id')

getOne(@Param('id') id: string) {

return this.products.find(p => p.id === +id);

}

@Post()

create(@Body() product: any) {

this.products.push({ id: Date.now(), ...product });

return { message: 'Product created successfully' };

}

@Put(':id')

update(@Param('id') id: string, @Body() updatedProduct: any) {

const index = this.products.findIndex(p => p.id === +id);

if (index === -1) return { message: 'Product not found' };

this.products[index] = { ...this.products[index], ...updatedProduct };

return { message: 'Product updated successfully' };

}

@Delete(':id')

delete(@Param('id') id: string) {

this.products = this.products.filter(p => p.id !== +id);

return { message: 'Product deleted successfully' };

}

}

**5. Crear un servicio**

El **servicio** contiene la lógica de negocio.  
Esto separa el controlador de la lógica interna.

nest g service products

Código de ejemplo:

// src/products/products.service.ts

import { Injectable } from '@nestjs/common';

@Injectable()

export class ProductsService {

private products = [];

findAll() {

return this.products;

}

findOne(id: number) {

return this.products.find(p => p.id === id);

}

create(product: any) {

const newProduct = { id: Date.now(), ...product };

this.products.push(newProduct);

return newProduct;

}

update(id: number, updatedProduct: any) {

const index = this.products.findIndex(p => p.id === id);

if (index === -1) return null;

this.products[index] = { ...this.products[index], ...updatedProduct };

return this.products[index];

}

delete(id: number) {

this.products = this.products.filter(p => p.id !== id);

}

}

Ahora, inyecta el servicio en el **controlador**:

// src/products/products.controller.ts

import { Controller, Get, Post, Put, Delete, Param, Body } from '@nestjs/common';

import { ProductsService } from './products.service';

@Controller('products')

export class ProductsController {

constructor(private readonly productsService: ProductsService) {}

@Get()

getAll() {

return this.productsService.findAll();

}

@Get(':id')

getOne(@Param('id') id: string) {

return this.productsService.findOne(+id);

}

@Post()

create(@Body() product: any) {

return this.productsService.create(product);

}

@Put(':id')

update(@Param('id') id: string, @Body() updatedProduct: any) {

return this.productsService.update(+id, updatedProduct);

}

@Delete(':id')

delete(@Param('id') id: string) {

this.productsService.delete(+id);

return { message: 'Product deleted successfully' };

}

}

**6. Conectar con Base de Datos**

Instala **TypeORM** y un cliente para tu BD.  
Por ejemplo, para **PostgreSQL**:

npm install @nestjs/typeorm typeorm pg

Configura la conexión en app.module.ts:

import { Module } from '@nestjs/common';

import { TypeOrmModule } from '@nestjs/typeorm';

import { ProductsModule } from './products/products.module';

@Module({

imports: [

TypeOrmModule.forRoot({

type: 'postgres',

host: 'localhost',

port: 5432,

username: 'postgres',

password: '12345',

database: 'mydb',

autoLoadEntities: true,

synchronize: true, // ❗ Solo en desarrollo

}),

ProductsModule,

],

})

export class AppModule {}

Crea una **entidad** para Product:

// src/products/product.entity.ts

import { Entity, PrimaryGeneratedColumn, Column } from 'typeorm';

@Entity()

export class Product {

@PrimaryGeneratedColumn()

id: number;

@Column()

name: string;

@Column('decimal')

price: number;

@Column({ default: true })

isActive: boolean;

}

Registra la entidad en el módulo:

// src/products/products.module.ts

import { Module } from '@nestjs/common';

import { TypeOrmModule } from '@nestjs/typeorm';

import { ProductsController } from './products.controller';

import { ProductsService } from './products.service';

import { Product } from './product.entity';

@Module({

imports: [TypeOrmModule.forFeature([Product])],

controllers: [ProductsController],

providers: [ProductsService],

})

export class ProductsModule {}

Usa el repositorio en el servicio:

// src/products/products.service.ts

import { Injectable } from '@nestjs/common';

import { InjectRepository } from '@nestjs/typeorm';

import { Repository } from 'typeorm';

import { Product } from './product.entity';

@Injectable()

export class ProductsService {

constructor(

@InjectRepository(Product) private productsRepo: Repository<Product>,

) {}

findAll() {

return this.productsRepo.find();

}

findOne(id: number) {

return this.productsRepo.findOne({ where: { id } });

}

create(product: Partial<Product>) {

const newProduct = this.productsRepo.create(product);

return this.productsRepo.save(newProduct);

}

async update(id: number, updatedProduct: Partial<Product>) {

await this.productsRepo.update(id, updatedProduct);

return this.findOne(id);

}

delete(id: number) {

return this.productsRepo.delete(id);

}

}

**7. Probar la API**

Usa herramientas como **Postman** o **cURL**:

* **GET**:  
  GET http://localhost:3000/products
* **POST**:  
  POST http://localhost:3000/products
* {
* "name": "Laptop",
* "price": 1200
* }
* **PUT**:  
  PUT http://localhost:3000/products/1
* {
* "price": 1300
* }
* **DELETE**:  
  DELETE http://localhost:3000/products/1

**8. Estructura final**

src/

├── products/

│ ├── product.entity.ts

│ ├── products.controller.ts

│ ├── products.module.ts

│ └── products.service.ts

├── app.module.ts

└── main.ts

**9. Buenas prácticas**

* Usar **DTOs** (Data Transfer Objects) con class-validator y class-transformer.
* Habilitar **Logs** y **Middleware** para manejo de errores.
* Separar entorno con **variables de entorno (.env)**.
* Desactivar synchronize: true en producción.
* Implementar **autenticación** con JWT (@nestjs/jwt).

¿Quieres que te haga un ejemplo con **DTOs** y validaciones para mejorar la API? 🚀